Porting C Programs

Chapter 1

Program Portability Considerations

This chapter describes the portability issues you must consider when migrating existing C source language programs to run on the VAX C compiler for ULTRIX systems (VAX C/ULTRIX). You may not need to migrate C programs. If this is not a concern to you, go to Chapter 2. Read this chapter for information about writing VAX C/ULTRIX programs for improved portability that might be useful in the future.

You will need this chapter if you fall into one of the following two groups:

- You are a user with source programs written for the portable c compiler (pcc) for ULTRIX
- You are a user with source programs written for the VAX C compiler for VMS (VAX C/VMS) Version 3.0 or higher

The first group of users is interested in the differences between the two compilers. The second group of users must learn about the changes imposed by both the compiler differences and the operating system differences. (The compiler differences are relatively minor.)

This chapter also discusses whether the source program, once modified to compile with VAX C on an ULTRIX system, can be recompiled and executed with its original compiler and operating system, if necessary.

This chapter does not describe how to migrate programs. Appendix C describes the most efficient methods for transporting the programs.

NOTE

> To distinguish when VAX C is running on the ULTRIX or the VMS operating system in this chapter, the following convention is adopted: the operating system name is appended after VAX C, as in VAX C/ULTRIX. The remainder of this manual omits the /ULTRIX designation, (unless it is needed for clarification) since the product being described is VAX C on the ULTRIX system.

This chapter presents the information based on the needs of the two specific groups of users. Refer to the sections corresponding to your portability requirements and study the descriptions that follow.

1.1 Differences Between pcc and VAX C/ULTRIX

VAX C is a C compiler available on VAX ULTRIX as vcc. pcc is the default C compiler on VAX ULTRIX. If you are an ULTRIX user and want to compile and link your existing pcc source programs with the VAX C/ULTRIX compiler, note that there are certain behavioral and linguistic differences between these two C compilers. By reading this section, you can decide if it will be easy or difficult to take the source program, once it is modified for VAX C/ULTRIX, and recompile it successfully on pcc.

1.1.1 Behavioral Differences

This section describes the following behavioral differences between the pcc and VAX C compilers:

- Number of passes and methods to perform preprocessing
- Preprocessor behavioral differences
- Optimization capabilities
- Object file formats
- Listing output
- Default options
- Unavailable options
- Unique options
- Compatibility with the lint utility
- Compiler error messages

One area of difference that does not exist concerns the use of system libraries. VAX C/ULTRIX supports all native ULTRIX system libraries, as does the portable C compiler (pcc). Thus, if you are migrating C programs from pcc to VAX C /ULTRIX, you do not need to change your system library calls.

1.1.1.1 Compiler Phases

The pcc compiler is both a multipass and a multiphase compiler. For example, the first phase of pcc invokes the C preprocessor (cpp) to output a temporary file that is used for compiling during the second phase. The VAX C/ULTRIX compiler, does not perform preprocessing in a separate phase. Instead, VAX C/ULTRIX is a single-phase compiler that integrates its preprocessing functions with its other functions in one pass. (This technique speeds up the compiler by eliminating the separate startup time for reloading a new image.)

As a single-phase, single-pass compiler, VAX C/ULTRIX does not produce assembly code as output, either by default or on request. This approach further increases the overall compilation speed. However, since pcc produces assembly code output in response to the **–S** option of the **cc** command, you may be used to generating assembly language code for subsequent editing. You cannot continue this practice when using the VAX C/ULTRIX compiler.

When you specify the **–E** option on the **vcc** command line, VAX C/ULTRIX preprocesses the file and produces source output. There is not a separate preprocessor; this capability is built into the VAX C/ULTRIX compiler. The cpp preprocessor is no longer invoked when the **–E** option is specified.

The **–Em** option invokes the cpp preprocessor, so that output is identical to that generated by the first phase of the pcc compiler. The **–Em** option generates information for the make utility.

Since VAX C is a one-pass compiler, it does not support forward references in either declarations or code. In pcc, an **extern** declaration of an object can be a forward reference to a later-declared static object. However, in VAX C, these attempts generate the warning message that there is a duplicate declaration. VAX C then takes steps to ensure that the two objects are distinct.

1.1.1.2 Preprocessor Behavioral Differences

Since VAX C/ULTRIX and pcc use different code for preprocessing, there are a few anomalies in behavior that may occur under certain conditions.

If the substitution text for a macro identifier also contains the name of that identifier, the VAX C/ULTRIX preprocessor recursively expands the identifier and enters an infinite loop. However, pcc's preprocessor does not do this. If a source program compiled with pcc causes the compiler to enter an infinite loop when migrated to VAX C/ULTRIX, examine the source code for this loop-inducing condition.

1.1.1.3 Optimization Capabilities

The VAX C/ULTRIX compiler employs certain global and many local (also known as peephole) optimizations to generate highly optimized code. As an example of a local optimization, the VAX C/ULTRIX compiler searches for certain combinations of multiple instructions that it can replace with single instructions. As an example of a global optimization, the VAX C/ULTRIX compiler searches for common subexpressions that can be consolidated.

These optimizations occur by default. However, when a VAX C/ULTRIX user compiles a program for debugging with the **–g** option to the **vcc** command, optimization is automatically disabled. Optimization can also be disabled by using **–V nooptimize**. (See Chapter 2.)

1.1.1.4 Object File Formats

In V4.0 of VAX C/ULTRIX, the vcc compiler now generates BSD .o format for its object files. This means that the ld linker can now be used to link object files generated by vcc. Consequently, files can be linked much faster, and standard ULTRIX utilities can now process VAX C/ULTRIX files.

The ld linker is the default linker for VAX C/ULTRIX Version 4.0 files, but ld will not link files produced by a version of vcc prior to Version 4.0, or files with a .obj extension.

The previous object file format can still be generated when you use the **–V lk_object** option on the **vcc** command. This specification causes the **vcc** shell to pass files to the lk linker instead of the default ld linker.

Both the **–V lk_object** and the **–V nolk_object** produce object files with .o extensions. Even though though both object file formats have the same extension, the ld linker will not link files produced with the **–V lk_object** option.

1.1.1.5 Listing Output

The VAX C/ULTRIX compiler can produce a listing that displays the source code, the symbol table, the machine code and cross-reference information, if you specifically request it through the **–v** option of the **vcc** command. (By default, no listing is output.) However, pcc cannot produce a similar listing, either by default or on request.

1.1.1.6 Default Options

There is only one difference in the options that are set by default for the pcc and the VAX C/ULTRIX compilers. With pcc, optimization is off by default; with VAX C/ULTRIX, optimization is on by default.

1.1.1.7 Unavailable Options

VAX C/ULTRIX does not support the following command-line options that are available with pcc:

Option	Meaning
–go	Generates symbol table information for sdb (obsolete).
–R	Makes initialized variables shared and read only (done in assembler).
–S	Generates an assembly language file that can be compiled with the assembler.

1.1.1.8 Unique Options

VAX C/ULTRIX offers some unique options at the command-line level. Among these are the aforementioned **–V standard=portable** option for reviewing portability and the **–v**"*filename*" option for generating listings.

When you invoke the VAX C/ULTRIX compiler, the option **–V standard=portable** is off by default so portability warnings are not automatically generated. However, you can request portability warnings by enabling this option. With pcc, the only way to obtain portability warnings is to invoke the lint utility.

In addition, pcc accepts input from standard input, which **vcc** does not. For more information about these options, see Chapter 2.

1.1.1.9 Compatibility with lint

If you use pcc and are used to using lint to check source programs, you may find that lint reports problems in VAX C source programs that are not real problems when the programs are compiled with VAX C/ULTRIX. For example, lint reports the use of the VAX C language extensions (which are explained in Sections 1.1.2.7 through 1.1.2.13) as problems.

As it compiles, VAX C/ULTRIX performs many of the same checks that lint provides, if you specify **–V standard=portable** on the **vcc** command line. Thus, VAX C/ULTRIX users may prefer not to use lint and to depend on the other methods for checking source programs that are provided by VAX C/ULTRIX. See Chapter 4 for more information about the alternatives to lint for VAX C/ULTRIX users.

1.1.1.10 Compiler Error Messages

The format of the error messages generated by VAX C/ULTRIX and pcc is very similar, but the contents are quite different. See the appropriate compiler documentation for assistance with the meanings of messages.

1.1.2 Language Differences

The language differences that exist between VAX C/ULTRIX and pcc have the following distinct origins which provide a convenient method of classification:

- VAX C includes some features from the draft proposed ANSI standard
- VAX C includes some of its own language extensions
- The compiler designers made different choices for certain similar capabilities or chose not to implement other features

The first two categories of differences show VAX C/ULTRIX to be a superset of the C language as implemented for pcc. The last category highlights a few incompatibilities. This section reviews all the differences in their related groups, since the implications for portability are consistent among the groups.

The primary language differences between pcc and VAX C/ULTRIX stem from the inclusion in VAX C of some additional features that are currently defined in the draft of a proposed ANSI standard for the C language. However, since this is a draft of a proposed standard and is subject to change, Digital reserves the right to change the VAX C language accordingly.

The items in the following list result from the VAX C/ULTRIX incorporation of the proposed ANSI standard features. Sections 1.1.2.1 through 1.1.2.6 describe them in more detail.

- Function prototypes
- Generic **void** pointers
- The **#pragma** preprocessor directive
- Hexadecimal characters in escape sequences
- Vacuous tag declarations
- Additional predefined macros
- The **const** and **volatile** modifiers

The following list summarizes the additional language differences due to the VAX C language extensions. Sections 1.1.2.7 through 1.1.2.13 describe the VAX C language extension differences in more detail.

- The **globaldef**, **globalref**, and **globalvalue** storage class specifiers
- The **noshare** and **readonly** storage class modifiers
- The **variant_struct** and **variant_union** data types
- The **_align** modifier
- The ability to address a constant in a function call
- Multicharacter constants
- The main_program option

There appear to be several differences presented in the two previous lists, but if you want to migrate a program from pcc to VAX C/ULTRIX you should not encounter difficulty, since these differences represent additional VAX C/ULTRIX capabilities. In fact, you may want to take advantage of these differences by recoding segments of existing pcc source programs for recompilation on VAX C/ULTRIX. These differences only represent restrictions to those contemplating running a VAX C/ULTRIX or VAX C/VMS source program on pcc. In that case, carefully rework the source code to remove all use of the features, or the program will fail to compile.

There are a few additional language-related differences that derive neither from VAX C extensions nor the proposed ANSI standard features, but require discussion. This category of differences (described in more detail starting with Section 1.1.2.14), presents the most serious difficulty for porting programs between pcc and VAX C/ULTRIX. These differences include the following:

- Specification of dollar signs ($) in identifiers
- Order of evaluation of subexpressions
- Initialization of external objects
- Compiler-generated global symbols
- The asm pseudo function call
- Variable initialization
- Functions which return a structure value
- Casts as lvalues

In spite of the incompatibility that these differences represent, they are very specific and so narrow in scope that you should rarely encounter them. Nevertheless, if a program incorporates one or more of these features, it may or may not successfully compile on both compilers or it may produce different results when run. Study these differences and then check your C programs for all possible instances, making necessary modifications before trying any program migration.

1.1.2.1 Function Prototypes

VAX C/ULTRIX permits function prototypes as described in Chapter 4, while pcc does not. Therefore, you should not specify function prototypes in your source code if you desire portability between the pcc and VAX C/ULTRIX compilers.

Function prototypes offer many important advantages. With function prototypes, the compiler can verify between definition and invocation whether the types of argument are assignment-compatible or whether different numbers of arguments exist. In this way, you can designate consistent typing of arguments. Some of the semantic checking that the VAX C compiler provides with function prototypes has traditionally been done on ULTRIX systems with the lint utility.

In the presence of a prototype, VAX C/ULTRIX may generate a different argument block based on the argument types specified in the prototype. The compiler can also provide optimizations based on the types of arguments in the argument block. VAX C/ULTRIX always promotes characters and short integers to integers. Since the type **float** passes a single-precision, floating-point argument and not a double-precision, floating-point argument, you must remember that on ULTRIX all math functions expect double-precision arguments to be passed.

Certain include files on VMS systems define functions using function prototypes. Their ULTRIX counterparts do not. Porting a C program from a VMS to an ULTRIX system, which depends on the type conversion implied by a function prototype on a VMS system may produce unexpected results on an ULTRIX system.

1.1.2.2 Generic Pointers

VAX C/ULTRIX permits the use of a generic pointer. The generic pointer is designated by **void ***, as defined in the draft proposed ANSI standard. For example, the following statement shows how you might use the generic pointer in a function prototype:

```
int memcpy (void *destination, void *source, int length);
```

In this case, the function memcpy is an object-copying function that takes three arguments. The first argument specifies the location that will receive the data, the second argument specifies the location of the data to be copied, and the third argument provides the number of bytes to be copied. The data types of the source and destination are not important to the operation of this function, but the VAX C compiler expects the types to be specified, and checks for compliance. To circumvent the compiler's typing requirements, the **void *** generic pointer is used in place of a data type specification for the first two arguments. It specifies that its associated argument is a pointer to data whose type can be arbitrary. Thus, arbitrary data types are successfully copied with this function.

See Chapter 7 for more information about the generic pointer.

This limitation presents no problems in migrating source programs from pcc to VAX C, but is identified here so you remain aware that using generic pointers in VAX C source programs prohibits their compilation by pcc.

1.1.2.3 The #pragma Preprocessor Directive

Another C language feature unique to VAX C/ULTRIX is the **#pragma** preprocessor directive. This directive allows VAX C/ULTRIX users to selectively enable or disable various default compiler behaviors. Source programs that include the **#pragma** directive do not successfully compile with pcc. See Chapter 9 for more information on **#pragma**.

1.1.2.4 Hexadecimal Characters in Escape Sequences

VAX C/ULTRIX allows the specification of hexadecimal characters in escape sequences; pcc does not. For more information, see Chapter 7.

1.1.2.5 Vacuous Tag Declarations

VAX C/ULTRIX allows the use of vacuous tag declarations that eliminate ambiguity in forward references to structure and union tags. For more information about vacuous tag declarations, see Chapter 7. Vacuous tag declarations are not permitted by pcc.

1.1.2.6 Additonal Predefined Macros

VAX C/ULTRIX allows the following macros, which are not recognized by pcc:

- __DATE__
- __TIME__

See Chapter 9 for more information about these predefined macros.

1.1.2.7 Storage-Class Specifiers

The **globaldef**, **globalref**, and **globalvalue** storage class specifiers are VAX C language extensions that are not available with pcc. See Chapter 8 for more information about storage class specifiers.

1.1.2.8 Storage Class Modifiers

The **noshare** and **readonly** storage class modifiers are VAX C language extensions that are not supported by pcc. See Chapter 8 for more information about storage-class modifiers. The **noshare** storage-class specifier is included in VAX C/ULTRIX only for reasons of compatibility with VAX C/VMS; it has no meaning in the ULTRIX implementation of VAX C.

1.1.2.9 The Variant Structure and Union Declarations

The **variant_struct** and **variant_union** declarations are VAX C language extensions that are not supported by pcc. See Chapter 7 for more information about these declarations.

1.1.2.10 The _align Modifier

The **_align** modifier, which allows you to align objects of any of the VAX C data types on a specified storage boundary, is a VAX C language extension that is not supported by pcc. See Chapter 8 for more information about this modifier.

1.1.2.11 The & Operator in Function Calls

Using the & operator with a constant in the argument list of a function call is allowed by VAX C, but is not supported by pcc.

1.1.2.12 Multicharacter Constants

Multicharacter constants are not allowed by pcc, but up to four characters can be specified in a character constant for VAX C/ULTRIX.

1.1.2.13 The main_program Option

The main_program option defines the main entry point in a program the same way that using the name main does. The main_program option provides a way to give the main function a different name. This feature is not supported by pcc.

See Chapter 4 for more information about the main_program option.

1.1.2.14 Specifying Dollar Signs ($) in Identifiers

Use of the dollar sign character ($) in identifiers is accepted by VAX C/ULTRIX. This is permitted by pcc, though the dollar sign character cannot be the first character of a macro name.

1.1.2.15 Order of Evaluation for Subexpressions

The C language does not define a precise order of evaluation for subexpressions found in either argument lists or general expressions. Thus, the pcc and VAX C compilers have each adopted different orders of evaluation. Subexpressions containing side-effect operators (such as ++ and −−) may produce different results with each compiler.

1.1.2.16 Initializing of External Objects

VAX C/ULTRIX will not allow initialization of the declaration of an external object if the declaration contains the **extern** keyword. With pcc, this notation is allowed.

1.1.2.17 Compiler-Generated Global Symbols

The following global symbols are implemented for pcc, but are not implemented with VAX C/ULTRIX:

- edata
- end
- etext

1.1.2.18 The asm Pseudo Function Call

The asm pseudo function call is allowed by pcc, but is not supported by VAX C/ULTRIX. VAX C/ULTRIX provides capabilities similar to asm through built-in functions; these provide access to directly access some VAX instructions from C code. Unlike asm, instead of these functions providing a string which contains an assembler instruction as the parameter, C variables and expressions are the parameters. For more information on these functions, see Chapter 10.

1.1.2.19 Variable Initialization

The following statement is a legal method of initializing the integer foo with pcc:

```
int foo 123;
```

This format is not accepted by VAX C/ULTRIX. The following example shows how the VAX C/ULTRIX implementation requires an equal sign (=) to perform the initialization:

```
int foo = 123;
```

1.1.2.20 Functions Which Return a Structure Value

VAX C/ULTRIX and pcc use different calling conventions to call a function that returns a structure. If you call a function that returns a structure from vcc and that function was compiled with pcc, the call will return unpredictable results. If you call a function that returns a structure from pcc and that function was compiled with vcc, a segmentation fault occurs.

1.1.2.21 Casts as lvalues

VAX C/ULTRIX allows casts to be used as lvalues, while the pcc compiler does not. The following statement is acceptable to vcc, but not pcc:

```
(* new_ptr_type) p = &q
```

1.2 Differences Between VAX C/VMS and VAX C/ULTRIX

If you want to compile and link your existing VAX C source programs with the VAX C/ULTRIX compiler, you must consider the minor behavioral and linguistic differences between the two VAX C compilers. You must learn how to compile, link, and run your source programs on ULTRIX. See Chapter 2 for more information on compiling and linking on ULTRIX.

The major differences discussed in this section can be categorized as follows:

- Language differences
- Include files
- Tool support
- Error message formats
- Structure alignment differences
- Behavioral differences
- Variable names
- The **#module** preprocessor directive

1.2.1 Language Differences

The VAX C/ULTRIX language is a major subset of the VAX C/VMS language, so there are only a few constructs that are not allowed in VAX C/ULTRIX programs.

1.2.1.1 CDD/Plus

If you want to migrate a VAX C program from VMS to ULTRIX, make sure that there are no references to the CDD/Plus data dictionary, which does not exist on ULTRIX systems. Search for all instances where **#dictionary** is specified and remove these references to CDD/Plus.

1.2.1.2 The #include Preprocessor Directive

Additional attention is required wherever the VAX C/VMS source program specifies the following preprocessor directive:

#include *identifier*

The specified identifier is subject to successful macro expansion. While VAX C/VMS allows all of the following possible resulting expansions, VAX C/ULTRIX permits only the specification of file paths in angle brackets (<>) or quotation marks ("):

- **#include** <file-spec>
- **#include** "file-spec"

- **#include** module-name

See Chapter 9 for more information regarding the **#include** preprocessor directive.

1.2.2 Include Files

VAX C/ULTRIX uses the include files provided on ULTRIX systems for pcc. There are differences between the include files on VMS and ULTRIX systems. Therefore, you should closely examine any include files on the system you are using. One significant difference is the **_tolower** and **_toupper** macros. On ULTRIX systems, the definitions add a constant value to their argument (the argument should be tested first with the **islower** and **isupper** macros before being passed to **_tolower** and **_toupper**). Also, the ULTRIX **_tolower** and **_toupper** macros can safely take arguments with side effects such as i++.

On VMS systems, the **_toupper** and **_tolower** macros test to make sure that the argument is in the appropriate range of letters before adding the constant value. The VMS versions of **_tolower** and **_toupper** cannot take arguments with side effects. The ULTRIX versions can.

1.2.3 Tool Support

The ULTRIX system does not provide interface support to two VMS software tools: Source Code Analyzer (SCA) and Language Sensitive Editor (LSE). Consequently, the compiler options **–V diagnostics** and **–V analyze_data** are not supported.

1.2.4 Error Message Formats

The format of the error messages differs between VMS and ULTRIX systems, but the compiler error message content is the same. Appendix B defines the VAX C/ULTRIX error messages.

1.2.5 Structure Alignment Differences

VAX C/VMS does not provide padding of structures and alignment of structure members by default; VAX C/ULTRIX does.

You can use the **#pragma [no]member_alignment** preprocessor directive in your programs to enable or disable the padding of structures and alignment of members at will. See Chapter 9 for more information about this directive.

1.2.6 Behavior Differences

Virtual address 0 is not a valid address on VMS systems. On ULTRIX systems, virtual address 0 is not guaranteed to contain any object. Therefore, a program that dereferences the null pointer (that is, a program that references virtual address 0) causes an access violation error on VMS systems but executes with undefined behavior on ULTRIX systems.

1.2.7 Variable Names

In VAX C/VMS, the maximum length of external variable names is 31. In VAX C/ULTRIX, the maximum length of external variable names is 255.

1.2.8 The #module Preprocessor Directive

The **#module** preprocessor directive is not implemented in VAX C/ULTRIX.